Tuning CUDA Applications for Pascal

Application Note

DA-08134-001_v11.4 | November 2021

Table of Contents

Chapter 1. Pascal Tuning Guide

1.1. NVIDIA Pascal Compute Architecture

Pascal is NVIDIA's latest architecture for CUDA compute applications. Pascal retains and extends the same CUDA programming model provided by previous NVIDIA architectures such as Kepler and Maxwell, and applications that follow the best practices for those architectures should typically see speedups on the Pascal architecture without any code changes. This guide summarizes the ways that an application can be fine-tuned to gain additional speedups by leveraging Pascal architectural features.[1]

Pascal architecture comprises two major variants: GP100 and GP104.[2] A detailed overview of the major improvements in GP100 and GP104 over earlier NVIDIA architectures are described in a pair of white papers entitled NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built for GP100 and NVIDIA GeForce GTX 1080: Gaming Perfected for GP104.

For further details on the programming features discussed in this guide, please refer to the CUDA C++ Programming Guide. Some of the Pascal features described in this guide are specific to either GP100 or GP104, as noted; if not specified, features apply to both Pascal variants.

1.2. CUDA Best Practices

The performance guidelines and best practices described in the CUDA C++ Programming Guide and the CUDA C++ Best Practices Guide apply to all CUDA-capable GPU architectures. Programmers must primarily focus on following those recommendations to achieve the best performance.

The high-priority recommendations from those guides are as follows:

- Find ways to parallelize sequential code,
- Minimize data transfers between the host and the device,
- Adjust kernel launch configuration to maximize device utilization,

[1] Throughout this guide, *Fermi* refers to devices of compute capability 2.x, *Kepler* refers to devices of compute capability 3.x, *Maxwell* refers to devices of compute capability 5.x, and *Pascal* refers to device of compute capability 6.x.
[2] The specific compute capabilities of GP100 and GP104 are 6.0 and 6.1, respectively. The GP102 architecture is similar to GP104.

- Ensure global memory accesses are coalesced,
- Minimize redundant accesses to global memory whenever possible,
- Avoid long sequences of diverged execution by threads within the same warp.

1.3. Application Compatibility

Before addressing specific performance tuning issues covered in this guide, refer to the Pascal Compatibility Guide for CUDA Applications to ensure that your application is compiled in a way that is compatible with Pascal.

1.4. Pascal Tuning

1.4.1. Streaming Multiprocessor

The Pascal Streaming Multiprocessor (SM) is in many respects similar to that of Maxwell. Pascal further improves the already excellent power efficiency provided by the Maxwell architecture through both an improved 16-nm FinFET manufacturing process and various architectural modifications.

1.4.1.1. Instruction Scheduling

Like Maxwell, Pascal employs a power-of-two number of CUDA Cores per partition. This simplifies scheduling compared to Kepler, since each of the SM's warp schedulers issue to a dedicated set of CUDA Cores equal to the warp width (32). Each warp scheduler still has the flexibility to dual-issue (such as issuing a math operation to a CUDA Core in the same cycle as a memory operation to a load/store unit), but single-issue is now sufficient to fully utilize all CUDA Cores.

GP100 and GP104 designs incorporate different numbers of CUDA Cores per SM. Like Maxwell, each GP104 SM provides four warp schedulers managing a total of 128 single-precision (FP32) and four double-precision (FP64) cores. A GP104 processor provides up to 20 SMs, and the similar GP102 design provides up to 30 SMs.

By contrast GP100 provides smaller but more numerous SMs. Each GP100 provides up to 60 SMs.[3] Each SM contains two warp schedulers managing a total of 64 FP32 and 32 FP64 cores. The resulting 2:1 ratio of FP32 to FP64 cores aligns well with GP100's new datapath configuration, allowing Pascal to process FP64 workloads more efficiently than Kepler GK210, the previous NVIDIA architecture to emphasize FP64 performance.

1.4.1.2. Occupancy

The maximum number of concurrent warps per SM remains the same as in Maxwell and Kepler (i.e., 64), and other factors influencing warp occupancy remain similar as well:

[3] The Tesla P100 has 56 SMs enabled.

- The register file size (64k 32-bit registers) is the same as that of Maxwell and Kepler GK110.
- The maximum registers per thread, 255, matches that of Kepler GK110 and Maxwell. As with previous architectures, experimentation should be used to determine the optimum balance of register spilling vs. occupancy, however.
- The maximum number of thread blocks per SM is 32, the same as Maxwell and an increase of 2x over Kepler. Compared to Kepler, Pascal should see an automatic occupancy improvement for kernels with thread blocks of 64 or fewer threads (shared memory and register file resource requirements permitting).
- Shared memory capacity per SM is 64KB for GP100 and 96KB for GP104. For comparison, Maxwell and Kepler GK210 provided 96KB and up to 112KB of shared memory, respectively. But each GP100 SM contains fewer CUDA Cores, so the shared memory available per core actually increases on GP100. The maximum shared memory per block remains limited at 48KB as with prior architectures (see Shared Memory Capacity).

As such, developers can expect similar occupancy as on Maxwell without changes to their application. As a result of scheduling improvements relative to Kepler, warp occupancy requirements (i.e., available parallelism) needed for maximum device utilization are generally reduced.

1.4.2. New Arithmetic Primitives

1.4.2.1. FP16 Arithmetic Support

Pascal provides improved FP16 support for applications, like deep learning, that are tolerant of low floating-point precision. The `half` type is used to represent FP16 values on the device. As with Kepler and Maxwell, FP16 storage can be used to reduce the required memory footprint and bandwidth compared to FP32 or FP64 storage. Pascal also adds support for native FP16 instructions. Peak FP16 throughput is attained by using a paired operation to perform two FP16 instructions per core simultaneously. To be eligible for the paired operation the operands must be stored in a `half2` vector type. GP100 and GP104 provide different FP16 throughputs. GP100, designed with training deep neural networks in mind, provides FP16 throughput up to 2x that of FP32 arithmetic. On GP104, FP16 throughput is lower, 1/64th that of FP32. However, compensating for reduced FP16 throughput, GP104 provides additional high-throughput INT8 support not available in GP100.

1.4.2.2. INT8 Dot Product

GP104 provides specialized instructions for two-way and four-way integer dot products. These are well suited for accelerating Deep Learning inference workloads. The `__dp4a` intrinsic computes a dot product of four 8-bit integers with accumulation into a 32-bit integer. Similarly, `__dp2a` performs a two-element dot product between two 16-bit integers in one vector, and two 8-bit integers in another with accumulation into a 32-bit integer. Both instructions offer a throughput equal to that of FP32 arithmetic.

1.4.3. Memory Throughput

1.4.3.1. High Bandwidth Memory 2 DRAM

GP100 uses High Bandwidth Memory 2 (HBM2) for its DRAM. HBM2 memories are stacked on a single silicon package along with the GPU die. This allows much wider interfaces at similar power compared to traditional GDDR technology. GP100 is linked to up to four stacks of HBM2 and uses two 512-bit memory controllers for each stack. The effective width of the memory bus is then 4096 bits, a significant increase over the 384 bits in GM200. This allows a tremendous boost in peak bandwidth even at reduced memory clocks. Thus, the GP100 equipped Tesla P100 has a peak bandwidth of 732 GB/s with a modest 715 MHz memory clock. DRAM access latencies remain similar to those observed on Maxwell.

In order to hide DRAM latencies at full HBM2 bandwidth, more memory accesses must be kept in flight compared to GPUs equipped with traditional GDDR5. Helpfully, the large complement of SMs in GP100 will typically boost the number of concurrent threads (and thus reads-in-flight) compared to previous architectures. Resource constrained kernels that are limited to low occupancy may benefit from increasing the number of concurrent memory accesses per thread.

Like Kepler GK210, the GP100 GPU's register files, shared memories, L1 and L2 caches, and DRAM are all protected by Single-Error Correct Double-Error Detect (SECDED) ECC code. When enabling ECC support on a Kepler GK210, the available DRAM would be reduced by 6.25% to allow for the storage of ECC bits. Fetching ECC bits for each memory transaction also reduced the effective bandwidth by approximately 20% compared to the same GPU with ECC disabled. HBM2 memories, on the other hand, provide dedicated ECC resources, allowing overhead-free ECC protection.[4]

1.4.3.2. Unified L1/Texture Cache

Like Maxwell, Pascal combines the functionality of the L1 and texture caches into a unified L1/Texture cache which acts as a coalescing buffer for memory accesses, gathering up the data requested by the threads of a warp prior to delivery of that data to the warp. This function previously was served by the separate L1 cache in Fermi and Kepler.

By default, GP100 caches global loads in the L1/Texture cache. In contrast, GP104 follows Kepler and Maxwell in caching global loads in L2 only, unless using the *LDG* read-only data cache mechanism introduced in Kepler. As with previous architectures, GP104 allows the developer to opt-in to caching all global loads in the unified L1/Texture cache by passing the `-Xptxas -dlcm=ca` flag to `nvcc` at compile time.

Kepler serviced loads at a granularity of 128B when L1 caching of global loads was enabled and 32B otherwise. On Pascal the data access unit is 32B regardless of whether global loads are cached in L1. So it is no longer necessary to turn off L1 caching in order to reduce wasted global memory transactions associated with uncoalesced accesses.

[4] As an exception, scattered writes to HBM2 see some overhead from ECC but much less than the overhead with similar access patterns on ECC-protected GDDR5 memory.

Unlike Maxwell but similar to Kepler, Pascal caches thread-local memory in the L1 cache. This can mitigate the cost of register spills compared to Maxwell. The balance of occupancy versus spilling should therefore be re-evaluated to ensure best performance.

Two new device attributes were added in CUDA Toolkit 6.0: `globalL1CacheSupported` and `localL1CacheSupported`. Developers who wish to have separately-tuned paths for various architecture generations can use these fields to simplify the path selection process.

Note: Enabling caching of globals in GP104 can affect occupancy. If per-thread-block SM resource usage would result in zero occupancy with caching enabled, the CUDA driver will override the caching selection to allow the kernel launch to succeed. This situation is reported by the profiler.

1.4.4. Atomic Memory Operations

Like Maxwell, Pascal provides native *shared* memory atomic operations for 32-bit integer arithmetic, along with native 32 or 64-bit compare-and-swap (CAS). Developers coming from Kepler, where shared memory atomics were implemented in software using a lock/update/unlock sequence, should see a large performance improvement particularly for heavily contended shared-memory atomics.

Pascal also extends atomic addition in global memory to function on FP64 data. The `atomicAdd()` function in CUDA has thus been generalized to support 32 and 64-bit integer and floating-point types. The rounding mode for all floating-point atomic operations is round-to-nearest-even in Pascal (in Kepler, FP32 atomic addition used round-to-zero). As in previous generations FP32 `atomicAdd()` flushes denormalized values to zero.

For GP100 atomic operations may target the memories of peer GPUs connected through NVLink. Peer-to-peer atomics over NVLink use the same API as atomics targeting global memory. GPUs connected via PCIE do not support this feature.

Pascal GPUs provide support system-wide atomic operations targeting *migratable allocations*[5] If system-wide atomic visibility is desired, operations targeting migratable memory must specify a system scope by using the `atomic[Op]_system()` intrinsics[6]. Using the device-scope atomics (e.g. `atomicAdd()`) on migratable memory remains valid, but enforces atomic visibility only within the local GPU.

Note: Given the potential for incorrect usage of atomic scopes, it is recommended that applications use a tool like CUDA memcheck to detect and eliminate errors.

As implemented for Pascal, system-wide atomics are intended to allow developers to experiment with enhanced memory models. They are implemented in software and some care is required to achieve good performance. When an atomic targets a migratable address backed by a remote memory space, the local processor page-faults so that the kernel can migrate the appropriate memory page to local memory. Then the usual hardware instructions are used to execute the atomic. Since the page is now locally resident, subsequent atomics

[5] Migratable, or *Unified Memory (UM)*, allocations are made with `cudaMallocManaged()` or, for systems with Heterogeneous Memory Management (HMM) support, `malloc()`.

[6] Here [Op] would be one of `Add`, `CAS`, etc.

from the same processor will not result in additional page-faults. However, atomic updates from different processors can incur frequent page-faults.

1.4.5. Shared Memory

1.4.5.1. Shared Memory Capacity

For Kepler, shared memory and the L1 cache shared the same on-chip storage. Maxwell and Pascal, by contrast, provide dedicated space to the shared memory of each SM, since the functionality of the L1 and texture caches have been merged. This increases the shared memory space available per SM as compared to Kepler: GP100 offers 64 KB shared memory per SM, and GP104 provides 96 KB per SM.

This presents several benefits to application developers:

- Algorithms with significant shared memory capacity requirements (e.g., radix sort) see an automatic 33% to 100% boost in capacity per SM on top of the aggregate boost from higher SM count.
- Applications no longer need to select a preference of the L1/shared split for optimal performance. For purposes of backward compatibility with Fermi and Kepler, applications may optionally continue to specify such a preference, but the preference will be ignored on Maxwell and Pascal.

Note: Thread-blocks remain limited to 48 KB of shared memory. For maximum flexibility, NVIDIA recommends that applications use at most 32 KB of shared memory in any one thread block. This would, for example, allow at least two thread blocks to fit per GP100 SM, or 3 thread blocks per GP104 SM.

1.4.5.2. Shared Memory Bandwidth

Kepler provided an optional 8-byte shared memory banking mode, which had the potential to increase shared memory bandwidth per SM for shared memory accesses of 8 or 16 bytes. However, applications could only benefit from this when storing these larger elements in shared memory (i.e., integers and fp32 values saw no benefit), and only when the developer explicitly opted in to the 8-byte bank mode via the API.

To simplify this, Pascal follows Maxwell in returning to fixed four-byte banks. This allows, all applications using shared memory to benefit from the higher bandwidth, without specifying any particular preference via the API.

1.4.6. Inter-GPU Communication

1.4.6.1. NVLink Interconnect

NVLink is NVIDIA's new high-speed data interconnect. NVLink can be used to significantly increase performance for both GPU-to-GPU communication and for GPU access to system memory. GP100 supports up to four NVLink connections with each connection carrying up to 40 GB/s of bi-directional bandwidth.

NVLink operates transparently within the existing CUDA model. Transfers between NVLink-connected endpoints are automatically routed through NVLink, rather than PCIe. The `cudaDeviceEnablePeerAccess()` API call remains necessary to enable direct transfers (over either PCIe or NVLink) between GPUs. The `cudaDeviceCanAccessPeer()` can be used to determine if peer access is possible between any pair of GPUs.

1.4.6.2. GPUDirect RDMA Bandwidth

GPUDirect RDMA allows third party devices such as network interface cards (NICs) to directly access GPU memory. This eliminates unnecessary copy buffers, lowers CPU overhead, and significantly decreases the latency of MPI send/receive messages from/to GPU memory. Pascal doubles the delivered RDMA bandwidth when reading data from the source GPU memory and writing to the target NIC memory over PCIe.

1.4.7. Compute Preemption

Compute Preemption is a new feature specific to GP100. Compute Preemption allows compute tasks running on the GPU to be interrupted at instruction-level granularity. The execution context (registers, shared memory, etc.) are swapped to GPU DRAM so that another application can be swapped in and run. Compute preemption offers two key advantages for developers:

- Long-running kernels no longer need to be broken up into small timeslices to avoid an unresponsive graphical user interface or kernel timeouts when a GPU is used simultaneously for compute and graphics.
- Interactive kernel debugging on a single-GPU system is now possible.

1.4.8. Unified Memory Improvements

Pascal offers new hardware capabilities to extend Unified Memory (UM) support. An extended 49-bit virtual addressing space allows Pascal GPUs to address the full 48-bit virtual address space of modern CPUs as well as the memories of all GPUs in the system through a single virtual address space, not limited by the physical memory sizes of any one processor. Pascal GPUs also support memory page faulting. Page faulting allows applications to access the same managed memory allocations from both host and device without explicit synchronization. It also removes the need for the CUDA runtime to pre-synchronize *all* managed memory allocations before each kernel launch. Instead, when a kernel accesses a non-resident memory page, it faults, and the page can be migrated to the GPU memory on-demand, or mapped into the GPU address space for access over PCIe/NVLink interfaces.

These features boost performance on Pascal for many typical UM workloads. In cases where the UM heuristics prove suboptimal, further tuning is possible through a set of migration hints that can be added to the source code.

On supporting operating system platforms, any memory allocated with the default OS allocator (for example, malloc or new) can be accessed from both GPU and CPU code using the same pointer. In fact, all system virtual memory can be accessed from the GPU. On such systems, there is no need to explicitly allocate managed memory using `cudaMallocManaged()`.

Appendix A. Revision History

Version 1.0

- Initial Public Release

Version 1.1

- Updated references to the CUDA C++ Programming Guide and CUDA C++ Best Practices Guide.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
www.nvidia.com